Adaptive Composition in Dynamic Service Environments

LINA BARAKAT, SIMON MILES, and MICHAEL LUCK
Department of Informatics, King's College London, UK, WC2R 2LS

Abstract

Due to distribution, participant autonomy and lack of local control, service-based systems operate in highly dynamic and uncertain environments. In the face of such dynamism and volatility, the ability to manage service changes and exceptions during composite service execution is a vital requirement. Most current adaptive composition approaches, however, fail to address service changes without causing undesirable disruptions in execution or considerably degrading the quality of the composite application. In response, this paper presents a novel adaptive execution approach, which efficiently handles service changes occurring at execution time, for both repair and optimisation purposes. The adaptation is performed as soon as possible and in parallel with the execution process, thus reducing interruption time, increasing the chance of a successful recovery, and producing the most optimal solution according to the current environment state. The effectiveness of the proposed approach is demonstrated both analytically and empirically through a case study evaluation applied in the framework of learning object composition.

Keywords: service composition, adaptive service execution, quality of service, request-based dominance

1. Introduction

Service-oriented computing (SOC) is a suitable paradigm for the sharing of resources and functionalities in large-scale open distributed environments (e.g. the web,
computational Grids, and peer-to-peer systems). In this paradigm, providers encapsulate their offerings, ranging from expensive hardware components to entire applications, within *services* and expose them through uniform, machine-readable interfaces (or metadata) on a network of customers. Via their accessibility, reusability, and loose coupling, services provide the building blocks for rapid and low-cost development of complex distributed applications spanning organisational boundaries. A key feature enabled by SOC is the dynamic binding mechanism. Based on this, a composite application (e.g. a business process, scientific workflow, or e-learning experience) can be structured as a collection of interdependent *abstract* tasks, with *concrete* services being selected for these tasks at run time according to service availability and specific user quality of service (QoS) needs, thus achieving great flexibility and personalisation.

Open distributed service-based systems, however, exhibit high degrees of dynamism and uncertainty for several reasons, either intentional or unintentional. Specifically, existing service providers, being autonomous and self-interested, may choose not to fulfil their promises (e.g. announce false capabilities to attract more customers), to upgrade/degrade their quality offerings (e.g. driven by competition), or to disconnect from the system at any time, while new providers could join instead. Even with long-standing and cooperative providers, availability and quality estimates of services could still frequently change due to other factors. For instance, a service’s response time could be significantly affected by the provider’s load and network traffic at that moment. Similarly, a service might suddenly become unavailable due to network/hardware failure.

Although the dynamic binding of services offers some tolerance against such dynamism and uncertainty, it does not guarantee the successful execution of the composite application, i.e. that the selected component services, for composition, behave as expected. This is because the selection step normally takes place before the start of execution, i.e. services are selected for *all* tasks in advance, to reason effectively about the satisfaction of global (application-level) quality criteria (e.g. total price and total time). Hence, changes to a selected service could occur at any time before the actual invocation of this service, especially when executing complex applications involving many tasks (the case with most realistic applications), or better services could emerge,
making the selected service combination no longer valid or no longer the best option.

Consequently, to accommodate service volatility, it is essential for the composite application to be equipped with adaptation capabilities at execution time. Ideally, such capabilities include the following goals: (G1) recovering from unexpected situations on their occurrence, (G2) exploiting new emerging opportunities to enhance the selected solution at any execution stage, (G3) proactively preventing future breaches and faulty behaviour by performing early corrective actions while appropriate alternatives are still available, (G4) producing an optimal solution for any instantiated service re-selection process, and (G5) keeping triggered adaptations transparent to the end user (i.e. eliminating execution interruption).

Existing approaches to adaptation usually achieve some of the above goals at the expense of others (see Figure 1 for a comparison). Specifically, current approaches (e.g. [1, 2, 3, 4, 5, 6]) can mostly be classified as reactive, performing corrective actions only after an exception has already occurred, thus lacking any ability to avoid erroneous behaviour or to improve performance, when possible. In addition, an interruption to the execution process is incurred until the corrective actions (usually through costly re-planning) are completed. Attempts to reduce such an interruption include applying fast heuristics (e.g. [4, 5]) or pre-computing backup plans beforehand (e.g. [2, 6]). While the former affects the solution quality, the constant changes of the service landscape may invalidate the latter, making the backups no longer applicable or poor-quality choices. Despite some recent efforts on proactive adaptation (e.g. [7, 8, 9]), they mostly focus on the early detection of exceptions, ignoring the actual adaptation process.

<table>
<thead>
<tr>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔️</td>
<td>X</td>
<td>✔️</td>
<td>X</td>
<td>partially</td>
</tr>
</tbody>
</table>

Reaction on violation:

| Execution-time heuristic re-selection | ✔️ | X  | X  | X  | partially |
| Execution-time optimal re-selection  | ✔️ | X  | ✔️ | X  | ✔️ |
| Pre-computed backups                | partially | X  | X  | X  | ✔️ |

Figure 1: Current adaptive composition approaches
In response, we build on our previous service selection model \[10, 11\], and equip it with an early, efficient, and optimality-retaining execution-time adaptive behaviour, capable of achieving all of the above goals during composite service execution, as summarised below.

**G1.** In situations where it is not possible to prevent undesired behaviour prior to its execution, our approach allows efficient recovery (with almost no interruption), yet effectively reasoning about the best forward replacement available.

**G2.** Whenever an optimisation opportunity is identified (e.g. due to the availability of new, better services), adaptation is triggered to improve the current solution, as opposed to existing approaches where the adaptation is mainly corrective.

**G3.** Reaction to changes is performed as soon as these occur in the environment. Hence, problems encountered in services scheduled for future execution are dealt with as early as possible, before reaching erroneous execution points where recovery opportunities are of lower quality or not possible.

**G4.** The combination of services produced by any triggered service re-selection process is always the best possible, given the tasks already executed and the current environment state.

**G5.** Adaptation transparency is achieved through: a parallel-to-execution reaction, triggering the adaptation process in parallel with the current component’s execution to maximise the chance of its completion before the next component’s invocation; a novel light reselection algorithm, applying efficient repair rules (originally introduced for selection-time reactivity \[11\]) to an already existing search graph in response to a change, thus facilitating a fast adaptation, especially in critical cases where the change occurs at a late stage or at the end of a component execution; and a novel prioritisation of changes, assessing the importance and urgency of each change encountered, and guiding the behavior of the executing system correspondingly (changes potentially affecting action points in the near future are handled urgently, while the adaptation to those of less importance is allowed to be carried out during the next component execution, without causing interruption).

The paper is organised as follows. Section \[2\] discusses related work. The basic service selection model is summarised in Section \[3\] followed by a motivating example.
in Section 4. A classification of changes is introduced in Section 5, based on which the adaptive behaviour of the system is analysed in Section 6, and an efficient re-selection algorithm is outlined in Section 7. Sections 8 and 9 provide a theoretical and empirical analysis, respectively, and Section 10 concludes the paper.

2. Related Work

Quality-based service selection has gained much attention from others. Like us, Yu et al. [12] and Li et al. [13] model it as a multi-constrained optimal path problem, and present heuristic algorithms to improve efficiency. In contrast, Canfora et al. [14] take a genetic algorithm approach. However, neither addresses adaptation to changes in a dynamic world.

To address the volatility of service environment, some efforts are aimed at fault avoidance, introducing preventive measures to reduce failures and quality deviations during execution, e.g. through redundant execution of services [15] or by providing accurate quality estimations [16]. Yet, since complete avoidance of execution-time exceptions is not possible, the ability to adapt to changes remains a critical requirement.

Many other efforts thus focus on achieving fault-tolerant behaviour to ensure that the system continues its intended execution, or at least terminates in a consistent state, in spite of the occurrence of failure or violation. In this regard, a number of approaches are concerned with incorporating exception handling mechanisms into the composition modeling language itself [17, 18], allowing the designer (or user) to control recovery actions at execution time. Although effective for specific exception types (e.g. invalid input/output parameters), language-integrated adaptation may not be suitable for some other types (e.g. additions, deletions, or changes in quality values of services). This is because such environment changes are difficult to predict by the designer, and would result in an explosion of the exception handler complexities. Therefore, in this paper, adaptation is achieved at the middleware level.

Satisfying particular transactional patterns by the composite service has also been proposed in order to increase composition reliability and fault tolerance at execution time [19, 20]. These efforts aim to minimise the risk for consumers by ensuring that the
execution terminates in a consistent state even when failures occur, achieved through
compensation policies allowing the effects of executed services to be undone. Such
approaches, however, offer rather extreme and costly exception-handling capabilities,
which may not be necessary in many situations, and are constrained to cooperative
environments. Nevertheless, accounting for transactional properties can be considered
an interesting extension to our approach.

A popular way of recovering from unexpected situations during execution (and
the closest to our work) is by triggering re-planning actions in response. Some such
efforts apply, during the re-planning stage, the same selection algorithm used to pro-
duce the initial solution, but incorporating the current execution status. For exam-
ple, Zeng et al. [21] recalculate assignments for the non-executed part of a work-
flow each time a change occurs during execution by adopting Integer Programming.

A re-planning triggering algorithm is introduced by Canfora et al. [1] to recalculate
quality values of a composite service according to the new information at execution
time (e.g., actual service qualities, or actual number of loop iterations), and if the new
qualities differ considerably from previously estimated ones, execution is stopped and
genetic-algorithm-based re-planning is triggered for remaining workflow tasks. A sim-
ilar execution-time re-planning approach, but based on Integer Programming, is pre-
vented by Ardagna et al. [3]. Others introduce heuristic methods for the re-selection
process to reduce its computational complexity. For example, Berbner et al. [4] use the
H1_RELAX_IP heuristic, backtracking on the results of a relaxed integer program, to
re-plan the remaining part of the workflow in a timely manner. Likewise, Lin et al. [5]
propose a region-based heuristic re-selection algorithm, which iteratively expands the
sub-process to be reconfigured until a satisfactory replacement is found. All these ap-
proaches can be categorised as reactive, performing corrective actions only after faulty
or quality-violating services are executed, thus ignoring emerging better opportunities,
lacking the ability to prevent erroneous behaviour (even when such behaviour can be
detected at an early stage), and causing an interruption to execution until re-selection
is performed. That is, as opposed to our work, these approaches fail to achieve goals
G2, G3, and G5.

In order to eliminate the undesired re-selection delay at execution time (goal G5),
some approaches (e.g. [2, 22, 6]) suggest supporting the composite application with pre-computed backup services to ensure its continuous execution without any extra delay in the face of component failures. However, the problem with such approaches is that, due to the dynamic nature of services, the backups produced during selection may no longer remain optimal, satisfactory, or even available during execution. As a result, the execution could be faced with either a low-quality alternative, or a costly re-planning process to achieve a successful (or better) recovery.

Finally, although there are recent attempts towards achieving proactive adaptation (i.e. to prevent future failure or improve performance), these are still very limited and mainly focus on the change detection part, giving little or no consideration to the actual adaptation process. Proposed proactive change detection methods include applying performance prediction techniques [8, 23, 9], testing the behaviour of services using generated test cases [7, 24], and subscribing to change requests with the registry [25]. Such detection efforts can be considered complementary to our work (which focuses instead on the latter change handling step). Like us, a few approaches also consider subsequent proactive adaptation actions (e.g. [8, 9, 26]), but these actions are mostly instantiated for corrective purposes, to prevent an anticipated problem, ignoring optimisation opportunities (G2). Furthermore, no proper modeling and management of the adaptation process, to avoid its interference with the application’s execution (G5), is provided. These issues are addressed in our approach, achieving all goals (G1..G5), as summarised in Section 1 and detailed below.

3. Basic Model

This section summarises the main components involved in the quality-based service selection problem, including our selection algorithm to solve this problem, originally introduced in [10]. See Figure 2 for the notation used.

3.1. Planning Knowledge Model

The planning knowledge for a particular objective can be represented as a task hierarchy \((T, tr, tf, tg)\), where: \(T\) is a finite set of the tasks involved; \(tr\) is the root
Figure 2: Sets and functions used throughout the paper

of the hierarchy (the goal task); \( t_f \) is a functionality description function, assigning to each task \( t \in T \) a semantic specification of its functional requirements; and finally \( t_g \) is a task decomposition function, which maps each non-leaf task \( t \in T \) to a set of directed acyclic graphs, each specifying a different way of decomposing \( t \) into finer-grained sub-tasks and their partial ordering constraints (execution order). An example planning knowledge for goal task plan holiday is shown in Figure 3

Note that task definition is kept generic to be applicable to a wide range of domains. It may refer, for example, to an operation signature (in terms of input and output parameters), to a resource specification, or simply to a term of an ontology agreed within a community. Moreover, different mechanisms are possible for discovering suitable (candidate) services for each task: by consulting a central service repository storing service metadata (e.g., a semantic search over SAWSDL\(^1\) descriptions of web services advertised in a UDDI registry); or by calling for service proposals over the network

\(^1\)http://www.w3.org/TR/sawSDL/
(e.g., using the contract net protocol [27]). We make no assumptions in our model about any specific technology or service discovery and matching mechanism, and leave this to the application domain, focusing instead on the generic problem of how to efficiently select and maintain the best combination of the available (discovered) services under the dynamism and uncertainty inherent in many such domains.

3.2. Service Model

The space of available services can be defined as a tuple, \((S, sf, sv)\), where: \(S\) is the set of all available services; \(sf\) is a functionality description function, which assigns to each service \(s \in S\) a semantic specification of its functionality, e.g. in OWL-S or WSDL-S; and finally \(sv\) is a quality of service (non-functional properties) specification function, which assigns to each service \(s \in S\) its value for a quality attribute \(a \in AN\) \((AN\) is the set of all quality attributes).

Based on this, the candidate services for task \(t \in T\), denoted \(cnd(t) \subset S\), are those services \(s \in S\) whose functional description, \(sf(s)\), semantically matches the functional requirements of task \(t\), \(tf(t)\).

3.3. Request Model

A composition request can be defined as a triple, \((rt, rc, rw)\). Task \(rt \in T\) is the goal task to be accomplished. Function \(rc\) represents the QoS constraints imposed for task \(rt\), and maps attribute \(a \in AN\) to an upper or lower user-defined bound for its value, depending on the attribute direction. That is, \(rc(a)\) is the minimum allowed value for attribute \(a\) if this attribute has an increasing direction (a higher value is better), or the maximum allowed value if attribute \(a\)’s direction is decreasing (a lower
value is better). For simplicity, henceforth we assume that all quality attributes are decreasing. Note that \( r_c(a) = \text{undf} \) in case of no restrictions on the value of attribute \( a \) by the user. Finally, function \( rw \) specifies the user’s preferences regarding different quality attributes, and assigns to each attribute \( a \in AN \), a user-defined weighting factor \( rw(a) \in [0, 1] \) reflecting its relative importance, s.t. \( \sum_{a \in AN} rw(a) = 1 \).

3.3.1. Request-based Selection Plans

Based on the planning knowledge model, multiple alternative abstract plans may be available for achieving the requested task \( rt \). These plans, denoted \( abspln(rt) \), correspond to all the possible expansions of the requested task, derived by recursively replacing task nodes with their decomposition graphs. For example, according to the planning knowledge hierarchy of Figure 3, task A has five possible abstract plans: \( plan1: \ A; \ plan2: \ B-C-D; \ plan3: \ G-C-D; \ plan4: \ B-C-E-F; \) and \( plan5: \ G-C-E-F. \) Yet, not all these plans are necessarily interesting with respect to the user request at hand. That is, a plan whose available instances are all guaranteed to violate the quality constraints can be filtered out from the planning search space of the current request without affecting the ability to find an optimal solution. Formally, given a user request, the abstract plans to be considered for the selection process, denoted \( SPLN \), are given as

\[
SPLN = \{ p \in abspln(rt) \mid \forall a \in AR, \text{aggr} \in \text{nodes}(p) tmn(t, a) \leq r_c(a) \}\] 

Here, \( AR \) is the set of constrained quality attributes; \( \text{nodes}(p) \) returns the task nodes of plan \( p \); \( tmn(t, a) \) associates task \( t \) with the best (minimum value) offered for attribute \( a \) by this task’s candidate services, i.e. \( \min_{s \in \text{cnd}(t)} (sv(s, a)) \); and \( \text{aggr} \) is some aggregation function that depends on the attribute considered. For example, possible aggregation functions for the quality attributes execution time, reliability, and throughput are the summation, multiplication, and minimum functions, respectively.

3.3.2. Request-based Non Dominated Services

The set of alternative composite services for achieving the requested task, denoted \( cmp(rt) \), is derived by instantiating plans \( SPLN \) (i.e. replacing the task nodes in each plan \( p \in SPLN \) with a particular combination of their candidate services). With the
increasing number of services per task, the number of such alternative compositions \(|cmp(rt)|\) can be exponential. However, this number could be reduced considerably by filtering out from the candidate space of each task, all the services uninteresting for the current request. Such uninteresting services are those request-based dominated by another candidate service for the same task, with a service \(s_j \in cnd(t)\) request-based dominating (r-dm) service \(s_i \in cnd(t)\) iff \(s_i\) is worse than \(s_j\) regarding all the constrained quality attributes \(AR\), and the overall utility value \(su\), i.e.

\[
\forall a \in AR, \ sv(s_i, a) \geq sv(s_j, a) \] \land \ [su(t, s_i) \leq su(t, s_j)] \land
\]

\[
\exists a \in AR, \ (sv(s_i, a) > sv(s_j, a)) \lor (su(t, s_i) < su(t, s_j))
\]

Here function \(su(t, s) \in [0, 1]\) returns the overall utility of service \(s \in cnd(t)\) regarding the user’s request, s.t. \(su(t, s) = \sum_{a \in AN} (rw(a) \cdot \frac{tmax(t, a) - sv(s, a)}{tmax(t, a) - tmin(t, a)})\), where \(tmax(t, a)\) is the maximum (and \(tmin(t, a)\) the minimum) value offered for attribute \(a\) by task \(t\)’s candidate services.

Request-based dominated services are not potential candidates for the optimal solution, and thus can be ignored when instantiating plans \(SPLN\).

### 3.4. Service Selection Problem

The service selection problem involves finding the best composite service to achieve the requested task, that both satisfies the user’s imposed quality constraints and maximises the overall utility with respect to user-defined quality weights.

The value offered by a composite service \(cs \in cmp(rt)\) for a particular quality attribute \(a\), \(cv(cs, a)\), is some aggregation \(aggr\) of the corresponding quality values for the component services, where \(aggr\) depends on the attribute considered. Based on this, the set of satisfactory composite services for the user’s request, can be defined as \(SCS = \{cs \in cmp(rt) | \forall a \in AN, (rc(a) \neq undf) \Rightarrow (cv(cs, a) \leq rc(a))\}\).

The solution composite service \(cs_{sol}\) for the user request is that satisfying: \(cs_{sol} \in SCS\) such that \(cu(cs_{sol}) = \max_{cs \in SCS} (cu(cs))\), where function \(cu(cs) \in [0, 1]\) represents the overall utility of composite service \(cs\), s.t. \(cu(cs) = \sum_a (rw(a) \cdot \frac{rmx(a) - cv(cs, a)}{rmx(a) - rmin(a)})\).

Here, \(rmx(a)\) returns the maximum (and \(rmin(a)\) the minimum) value offered for attribute \(a\) by the requested task’s actual plans (these maximum/minimum values can
be estimated by aggregating for each abstract plan the $tmx/tmn$ values of its tasks, and then calculating the maximum/minimum of these aggregated values).

3.5. Service Selection Algorithm

We model the service selection problem as a multi-constrained optimal path selection problem in a directed graph, called the plan paths graph $(V_{PK}, E_{PK})$, where each path corresponds to an alternative abstract plan for achieving the requested task. Note that we assume all abstract plans have a sequential structure (other structures can be transformed to the sequential structure using existing techniques [28]). Figure 4 provides the plan paths graph for the planning knowledge of Figure 3.

Based on the multi-constrained Bellman-Ford algorithm [29], our service selection algorithm is as follows. Each node $v$ in the plan paths graph stores the optimal instances, denoted as $oi(v, p_v)$, for each path $p_v + v$ discovered so far from the start node to $v$ (an instance of a path is a possible replacement of its task nodes with candidate services). In order to maximise utility, the concept of optimal paths in the original Bellman-Ford algorithm is updated so that an instance of path $p$ is considered optimal if no other possible instance of the same path has both better values for all the constrained attributes and better utility. Moreover, to reduce the number of optimal instances, only those satisfying the quality constraints are maintained in each node. After traversing all graph nodes in topological order, the solution is the optimal composite service that has the best utility at the destination node.

In order to ensure that only plans $SPLN$ are considered during selection, each node $v$ in the plan paths graph is associated with the set of its valid predecessors $vldprd(v)$, which can be defined as follows: given a path $p_v + v$ from the start node to $v$, path $p_v$ is considered a valid predecessor of node $v$ if there exists at least one path $p_i$ from $v$ to the
destination node, such that \( p_v + p_i \) is a satisfactory abstract plan, i.e. \( p_v + p_i \in SPLN \).

Based on this, when processing an edge \((u, v)\), only the optimal composite services stored in \( u \) that are instances of \( v \)'s valid predecessors are considered. More details on this selection algorithm can be found in [10].

### 4. Example of Service Changes

Consider an example in which the user has issued a request to achieve task \( A \), and is interested in minimising price (pr) while satisfying the constraint that execution time (ex) is less than 100. The plan paths graph for the requested task, and the request-based non-dominated candidate services of the sub-tasks involved are depicted in Figures 4 and 5 respectively. Based on this, the set of plans to be considered for selection \( SPLN = \{BCEF\} \) (plans \( A, BCD, GCD, \) and \( GCEF \) are excluded since tasks \( A \) and \( D \) do not have any available services, and all the available instances of plan \( GCEF \) violate the execution time constraint). Given set \( SPLN \), the valid predecessors of the nodes are as shown in Figure 5. The optimal solution for the user is instance \( s_{B1}s_{C2}s_{E2}s_{F1} \) (ex:95, pr:90) which has the lowest price. In what follows, we give three example scenarios of service changes during the execution of the selected composite service \( s_{B1}s_{C2}s_{E2}s_{F1} \).

**Scenario 1:** While executing service \( s_{B1} \) of composite service \( s_{B1}s_{C2}s_{E2}s_{F1} \), a new service \( s_{D1} \) (ex:40, pr:10) joins the candidate services of task \( D \). As a result, plan \( BCD \) is added to set \( SPLN \), and two additional instances of this plan are satisfactory from this point, of which composite service \( s_{B1}s_{C2}s_{D1} \) (ex:90, pr:70) is better than the selected composition \( s_{B1}s_{C2}s_{E2}s_{F1} \) for both price and execution time.

**Scenario 2:** While executing service \( s_{B1} \) of composite service \( s_{B1}s_{C2}s_{E2}s_{F1} \), service \( s_{C2} \) becomes unavailable. Here, simply replacing \( s_{C2} \) with \( s_{C1} \) will result in composi-
tion $s_{B1} s_{C1} s_{E2} s_{F1}$ (ex:80, pr:110), which is not optimal regarding price. Hence, both $s_{C2}$ and $s_{E2}$ should be substituted in this case in order to obtain the new optimal satisfactory solution, which is service $s_{B1} s_{C1} s_{E1} s_{F1}$ (ex:92, pr:95).

**Scenario 3:** While executing service $s_{B1}$ of composite service $s_{B1} s_{C2} s_{E2} s_{F1}$, service $s_{C2}$ changes its quality values to (ex:50, pr:20). From this point, the selected composite service is no longer satisfactory, and the new optimal satisfactory one is $s_{B1} s_{C1} s_{E1} s_{F1}$ (ex:92, pr:95).

5. **Service Change Categorisation**

As illustrated above, a change to the service landscape during execution may cause corresponding changes in the optimal composite services possible from that point (and potentially affecting the best solution), thus necessitating their recalculation in response (which we refer to as the re-selection process). Generally, the importance and urgency of responding to an encountered service change, i.e. triggering the re-selection process, vary depending on whether this change affects the non-dominated services of the respective task and other factors. Based on this, we propose categorising execution-time service changes into changes not to be considered and changes to be considered. These categories are detailed next after modelling the effect of a service change on the request-based non-dominated services of the task affected. In what follows, $\alpha_o$ and $\alpha_n$ represent $\alpha$ before and after a change occurrence.

5.1. The effect on non-dominated services

A change to the available services of task $t_{ch} \in V_{PK}$, might affect this task’s set of request-based non-dominated services $rcnd(t_{ch})$, causing the addition of new services $AD$ while removing existing ones $RM$, i.e. $rcnd_n(t_{ch}) = (rcnd_o(t_{ch}) \setminus RM) \cup AD$. The definition of sets $AD$ and $RM$ depends on the type of change that occurred, as detailed next.

5.1.1. Addition of a Service

Where a new service $s_n$ joins the candidate services of task $t_{ch}$, i.e. $cnd_n(t_{ch}) = cnd_o(t_{ch}) \cup \{s_n\}$, two cases are distinguished. If $\exists s \in rcnd_o(t_{ch})$ s.t. $s \text{ r-dm } s_n$,
no change is made to set $rcnd_o(t_{ch})$, i.e. $AD = RM = \emptyset$. Otherwise, service $s_n$ is added to set $rcnd_o(t_{ch})$, i.e. $AD = \{s_n\}$, removing from this set all the services request-based dominated by $s_n$, i.e. $RM = \{s \in rcnd_o(t_{ch}) \mid s \text{ r-dm } s\}$.

5.1.2. Deletion of a Service

Where a candidate service $s_o$ of task $t_{ch}$ becomes unavailable, i.e. $cnd_n(t_{ch}) = cnd_o(t_{ch}) \setminus \{s_o\}$, the following two cases are distinguished. If service $s_o$ is not a member of set $rcnd_o(t_{ch})$, its deletion does not affect this set, i.e. $AD = RM = \emptyset$. Otherwise, $s_o$ is removed from $rcnd_o(t_{ch})$, i.e. $RM = \{s_o\}$, adding to it all task $t_{ch}$’s candidate services not previously included in this set which, as a result of eliminating $s_o$, become non-dominated according to the current request, i.e. $AD = \{s \in srdd(s_o) \mid \forall s_i \in (rcnd_o(t_{ch}) \setminus \{s_o\}) \cup srdd(s_o), \neg(s_i \text{ r-dm } s)\}$, where $srdd(s_o) = \{s_i \in cnd_n(t_{ch}) \mid s_o \text{ r-dm } s_i\}$ is the set of candidate services request-based dominated by service $s_o$.

5.1.3. Changes in the Quality Values of a Service

Where a candidate service $s_o$ of task $t_{ch}$ changes its quality values, i.e. $cnd_n(t_{ch}) = (cnd_o(t_{ch}) \setminus \{s_o\}) \cup \{s_{ch}\}$, with $s_{ch}$ denoting this service after the change, the following two cases are distinguished. Case 1: $s_o \notin rcnd_o(t_{ch})$. Here, if service $s_o$ r-dm $s_{ch}$, no change to set $rcnd_o(t_{ch})$ is required, i.e. $AD = RM = \emptyset$. Otherwise, this case is treated similarly to the addition of a new service $s_n = s_{ch}$. Case 2: $s_o \in rcnd_o(t_{ch})$. Here, we have the following three sub-cases. Case 2.1: $s_{ch}$ r-dm $s_o$. In this case, service $s_o$ is replaced with service $s_{ch}$, removing from $rcnd_o(t_{ch})$ all the services that are request-based dominated by $s_{ch}$, i.e. $AD = \{s_{ch}\}$, $RM = \{s_o\} \cup \{s \in rcnd_o(t_{ch}) \setminus \{s_o\}, s_{ch} \text{ r-dm } s\}$. Case 2.2: $s_o$ r-dm $s_{ch}$. In this case, sets $AD$ and $RM$ are similar to those in the deletion case, i.e. $AD = \{s \in srdd(s_o) \mid \forall s_i \in (rcnd_o(t_{ch}) \setminus \{s_o\}) \cup srdd(s_o), \neg(s_i \text{ r-dm } s)\}$, $RM = \{s_o\}$ (notice that $s_{ch} \in srdd(s_o)$ in this case). Case 2.3: $s_{ch}$ and $s_o$ are incomparable according to the current request. In this case, if $\exists s \in rcnd_o(t_{ch}) \setminus \{s_o\}$ s.t. $s$ r-dm $s_{ch}$, then sets $AD$ and $RM$ are defined as in Case 2.2. Otherwise, the services to be added are service $s_{ch}$ along with all task $t_{ch}$’s candidate services not previously included in
which, as a result of replacing \( s_o \) with \( s_{ch} \), become non-dominated according to the current request, i.e. \( AD = \{ s_{ch} \} \cup \{ s \in srdd(s_o) \mid \forall s_i \in (rcnd_o(t_{ch}) \setminus \{ s_o \}) \cup \{ s_{ch} \} \cup srdd(s_o), \neg(r-dm s) \} \). The services to be removed are service \( s_o \) plus all the services in \( rcnd_o(t_{ch}) \setminus \{ s_o \} \) that are request-based dominated by \( s_{ch} \), i.e. \( RM = \{ s_o \} \cup \{ s \in rcnd_o(t_{ch}) \setminus \{ s_o \}, s_{ch} r-dm s \} \).

5.2. Changes not to be considered

A change to the available services of task \( t_{ch} \) while executing task \( t_{inv} \) (the task currently invoked of the selected solution) need not be considered, i.e. does not trigger the re-selection process, \textbf{iff} one of the following is satisfied: task \( t_{ch} \) is already executed, i.e. \( t_{ch} = t_{inv} \) or \( t_{ch} \) appears before \( t_{inv} \) according to the topological order of the plan paths graph; task \( t_{ch} \) is not part of the plan being executed \( (p_{sel}) \) and does not belong to any satisfactory plan after the change, i.e. \( (t_{ch} \notin nodes(p_{sel})) \land (vldprd_n(t_{ch}) = \emptyset) \); or the request-based non-dominated services of task \( t_{ch} \) are not affected by the change, i.e. \( AD = RM = \emptyset \).

5.3. Changes to be considered

A change to the available services of task \( t_{ch} \) while executing task \( t_{inv} \) needs to be considered, i.e. triggers the re-selection process, \textbf{iff} all of the following are satisfied: task \( t_{ch} \) is not executed yet, i.e. \( t_{ch} \) appears after \( t_{inv} \) according to the topological order of the plan paths graph; \( t_{ch} \) belongs to the plan being executed \( (p_{sel}) \) or belongs to at least one satisfactory plan after the change, i.e. \( (t_{ch} \in nodes(p_{sel})) \lor (vldprd_n(t_{ch}) \neq \emptyset) \); and the request-based non-dominated services of task \( t_{ch} \) are affected by the change, i.e. \( (AD \neq \emptyset) \lor (RM \neq \emptyset) \). Changes to be considered are further divided into non-affecting changes and affecting changes, as detailed next.

5.3.1. Non-affecting changes

A change is non-affecting if it has an impact on the optimal composite services possible from that point, but does not affect the best solution (the need to respond to this category of change is justified in Section\textsuperscript{7}). Having no effect on the best solution, this category of change does not cause any delay to the execution process. In other words,
the solution composite service can continue its execution even if the re-selection process in response to the change is still running. Generally, a change to be considered is regarded as non-affecting in the following cases: the deletion of a non-selected service (a service that is not part of the current best solution); and changes in the quality values of a non-selected service \( s_o \) (\( s_{ch} \) denotes this service after the change) such that \((s_o \text{ r-dm } s_{ch}) \text{ or } ((s_o \text{ is incomparable with } s_{ch}) \text{ and } (s_{ch} \notin AD))\).

5.3.2. Affecting changes

A change is affecting if it has an impact on the optimal composite services possible from that point, and could cause a modification to the best solution. This category is further divided into non-interrupting changes and interrupting changes.

Non-interrupting changes are those affecting changes, the reaction to which does not cause any interruption between service executions, since the next service to be executed can be identified without the need for re-selection to be completed. Specifically, an affecting change to the services of task \( t_{ch} \) is non-interrupting \textbf{iff} task \( t_{ch} \) is the next task to be executed according to the current solution, with service \( s_{sel} \) being the currently selected service for this task, and \textit{one} of the following is satisfied: the change that occurred is the addition of a new service \( s_n \) such that \( s_n \text{ r-dm } s_{sel} \); or the change is a modification in the quality values of service \( s_o \) (\( s_{ch} \) denotes this service after the change) such that \( s_{ch} \text{ r-dm } s_{sel} \) (note that \( s_{sel} \) might be the service affected by the change, i.e. \( s_o = s_{sel} \)). Intuitively, responding to such changes will result in replacing service \( s_{sel} \) with service \( s_n \) (in the addition case), and with service \( s_{ch} \) (in the modification case). Hence, the next service can be anticipated without requiring interruption.

Interrupting changes are those affecting changes, the reaction to which might result in an interruption to the composite service execution. This is because the next service to be executed cannot be identified prior to performing re-selection, thus causing the execution process to stop until re-selection is completed. Specifically, an affecting change to the services of task \( t_{ch} \) is interrupting in the following cases. \textit{Case 1:} the addition of a new service \( s_n \) such that \textit{one} of following is satisfied: \( t_{ch} \) is not part of the plan being executed; \( t_{ch} \) belongs to the plan being executed and \( s_n \) is incomparable with
Case 2: the deletion of a selected service. Case 3: changes in the quality values of a non-selected service \( s_{\text{ch}} \) (denotes this service after the change) such that all of the following are satisfied: \([s_{\text{ch}} \rdfs s] \lor [(s_{\text{ch}} \text{ is incomparable with } s_a) \land (s_{\text{ch}} \in AD)]; \) and \([t_{\text{ch}} \text{ is not the next task to be executed}] \lor \lnot [s_{\text{ch}} \rdfs s_{\text{sel}}], \) where \( s_{\text{sel}} \) is the currently selected service for task \( t_{\text{ch}}. \) Case 4: changes in the quality values of a selected service \( s_{\text{sel}} \) (denotes this service after the change) such that \( t_{\text{ch}} \) is not the next task to be executed or \( \lnot (s_{\text{ch}} \rdfs s_{\text{sel}}). \) Note that all the three change scenarios in Section 4 are considered interrupting, satisfying Case 1, Case 2, and Case 4, respectively.

6. Adaptive Execution Behaviour

Delaying the re-selection process until a violating behaviour is invoked results in undesired effects at execution time. For instance, observing the unavailability of service \( s_{C_2} \) in Scenario 2 only when trying to invoke this service causes execution to stop until re-selection is performed. Similarly, in Scenario 3 detecting the changes in the quality values of service \( s_{C_2} \) only after its execution results in an unrecoverable situation, since no satisfactory solution can be found from this point (all the service combinations including services \( s_{B_1} \) and \( s_{C_2} \) violate the user’s execution time constraint).

To tackle this, we propose an early, parallel-to-execution adaptive system behaviour, where adaptation to changes is performed as soon as these changes occur in the environment, concurrently with the execution of the current service, thus reducing the delay between service executions, and increasing the chance of a successful recovery. For instance, in Scenario 2 re-selecting services for tasks \( C, E \) and \( F \) in response to the deletion of service \( s_{C_2} \) can be achieved while executing service \( s_{B_1}, \) without causing extra delay.

Based on the change categories introduced, such an adaptive execution behaviour can be modelled using the finite state automaton in Figure 6 which consists of five states. States \( ex - \alpha \) indicate that a component service of the best solution is currently
running and, at the same time, the following is satisfied according to the value of $\alpha$:

when $\alpha = nch$, no re-selection is being performed by the system; when $\alpha = naff$, a re-selection is being performed in response to a set of non-affecting changes; when $\alpha = nint$, a re-selection is being performed in response to a set of changes including at least one non-interrupting change and no interrupting changes; and finally, when $\alpha = int$, a re-selection is being performed in response to a set of changes including at least one interrupting change. State $nex$ indicates that the best solution execution is currently interrupted until re-selection is completed.

The behaviour of the system is interpreted as follows. The execution begins in state $ex-nch$, by invoking the first component service. With the occurrence of a change to be considered during a component execution, the system transitions to one of the states $ex-naff$, $ex-nint$, or $ex-int$, based on the change category, which could be a non-affecting change (event $naff$), an affecting and non-interrupting change (event $nint$), or an affecting and interrupting change (event $int$). The change category is identified with respect to the currently selected solution (we assume that the time required for this identification is negligible, especially when compared to re-selection time). The
selected solution may be updated each time a re-selection is completed (event \(e-ch\)), causing the system to return to state \(ex-nch\).

After the successful execution of a component service (event \(e-ex\)), the state into which the system transitions is determined based on its current state, as follows. If the system is in state \(ex-nch\), i.e. no re-selection is being performed, the next service in the currently selected solution is invoked, without changing the state of the system. If the system is in state \(ex-naff\), i.e. the re-selection being performed will not affect the currently selected solution, the next service in this solution is invoked, without changing the state of the system. In other words, the re-selection is carried on while executing the next service. If the system is in state \(ex-int\), i.e. the next service to be executed cannot be identified before completing the re-selection being performed, the system transitions to state \(nex\), and remains in this state until re-selection is completed and the next service to be invoked is determined. Finally, if the system is in state \(ex-nint\), two cases are distinguished according to set \(srds(s_{nxt})\), the set of services dominating the currently selected service \(s_{nxt}\) for the next task in the execution order, among the request-based non-dominated services of this task. Case 1: \(|srds(s_{nxt})| = 1\), in which the next service to be executed can be estimated without the need for re-selection to be completed. This service, \(s_{nxt-new} \in srds(s_{nxt})\), is thus invoked without delaying execution, causing the system to transition to state \(ex-int\). In other words, the re-selection is continued while executing service \(s_{nxt-new}\), but is considered interrupting since the next service to invoke after service \(s_{nxt-new}\) cannot be known prior to completing re-selection. Case 2: \(|srds(s_{nxt})| \neq 1\), in which the next service to be executed cannot be determined before the re-selection is completed. Therefore, the execution process is interrupted by moving to state \(nex\) to continue the re-selection.

The case where the current component service delivers unexpected quality values (event \(e-ex-v\)) is considered an interrupting change, and thus also causes the system to enter state \(nex\), regardless of its current state.

**Example.** Consider Scenario \(2\) of Section \(4\) with the initial solution \(s_B1s_C2s_E2s_F1\). Invoking \(s_B1\) initiates the adaptive execution behaviour at state \(ex-nch\). Since the deletion of \(s_C2\) while executing \(s_B1\) is an interrupting change, it triggers the transition of the system to state \(ex-int\) to indicate a running re-selection. Once the re-
selection is completed, the system goes back to state ex-nch, updating the solution to $s_{B1}s_{C1}s_{E1}s_{F1}$.

7. Efficient Service Reselection

The adaptive behaviour proposed above triggers re-selection in response to changes in parallel with execution, in order to avoid extra delays. However, a costly re-selection process could still cause an interruption to execution, especially if the change is only discovered at a late stage or (in the worst case) at the end of the current component execution. In response, we introduce here a light re-selection approach, applying efficient repair rules to an already existing search graph (the graph produced by the initial selection process), without expensive recalculations from scratch. The idea is to apply the selection algorithm (of Section 3.5) prior to execution, in order to generate the search graph (i.e. generate the optimal instances for each task node), and to select the initial solution. The search graph is then kept valid during execution by continuously adjusting it with respect to the environment state (which justifies the need to account for non-affecting changes). Maintaining the graph validity ensures that, whenever any change occurs (especially a critical, affecting one), only a minimal number of modifications to the affected part of the graph are required in response, thus increasing the chance that the adaptation to the change terminates before the end of the current component execution. Next, we first introduce the search graph enabling efficient execution-time adaptivity, followed by the graph repair rules in response to a change (i.e. the re-selection algorithm).

7.1. Execution-time Search Graph

In the simplest case, with no changes encountered, the validity of the search graph should be maintained against the execution progress of the selected solution. This, however, could be costly to achieve with the forward version of the plan paths graph, where the task nodes are processed (by the selection algorithm) according to their execution order. To illustrate, consider the example of Section 4 with the forward plan paths graph in Figure 4 and the initial solution $s_{B1}s_{C2}s_{E2}s_{F1}$. Once service $s_{B1}$
is invoked, the optimal instances recorded at the remaining, non-executed task nodes (i.e., tasks nodes C, D, E, and F) are no longer valid. This is because these instances (which correspond to paths beginning with node B) account for service $s_{B2}$ as a possible candidate for executing task B, no longer the case after $s_{B1}$’s execution.

To tackle this, we apply the selection algorithm on the reverse version of the plan paths graph, generated by reversing the direction of edges in the original plan paths graph (i.e., the start node of the reverse plan paths graph is the end node of the original one). For example, the reverse graph for the plan paths graph of Figure 4 is provided in Figure 7. Such modified selection produces the same best solution, while maintaining the validity of optimal instances at the non-executed task nodes as the execution advances, due to the reverse processing order of nodes. For instance, in our example, when selection is performed on the reverse graph of Figure 7, executing service $s_{B1}$ does not affect the optimal instances of task node C which, in this case, records instances of paths DC and FEC. The same holds for the other, non-executed task nodes D, E and F. Only the optimal instances of the node executed, B, are affected, and would need to be re-computed if a change occurs (see Section 7.2).

Based on this, each time a new component service, $s_{inv}$, of the selected solution, $ins_{ex} + s_{inv} + ins_{uex}$, is invoked (where $ins_{ex}$ and $ins_{uex}$ correspond to the executed and non-executed parts of this solution, respectively), only the following adjustments are required to the search graph: changing the destination node to task $t_{inv}$ (the task being executed), with its request-based non-dominated services being set to instance $ins_{ex} + s_{inv}$; and updating the valid predecessors of nodes through adjusting the selection plans $SPLN$, such that all the plans not beginning with the already executed path are removed from set $SPLN$, since these are invalid from this point. In our example,
reflecting the execution of service \( s_{B1} \), on the reverse search graph, would only involve changing the destination node to node \( B \), and adjusting \( rcnd(B) \) to \( rcnd(B) = \{ s_{B1} \} \).

Here, no modification is required to set \( SPLN \) (and consequently the valid predecessors) since only plan \( BCEF \), currently under execution, is included in this set.

### 7.2. Search Graph Repair Rules

To account for a service change to be considered at task node \( t_{ch} \), while executing service \( s_{inv} \) (of task \( t_{inv} \)), we apply only the necessary updates to the optimal instances at the nodes of the reverse search graph, without re-computing these instances from scratch. This is achieved by associating each valid predecessor \( p_u \), of each node \( u \), with three mutable sets capturing the updates required: an additional services set \( as(u, p_u) \subset rcnd(u) \) specifying what services of node \( u \) need to be joined with path \( p_u \)'s optimal instances when updating the optimal instances of path \( p_u + u \); an additional instances set \( ai(u, p_u, i \in \mathbb{Z}^+) \subset S \) specifying what optimal instances of path \( p_u \) need to be joined with node \( u \)'s services when updating the optimal instances of path \( p_u + u \) (i.e. \( s \in ai(u, p_u, i) \) indicates that, of the additional optimal instances \( ins \) of path \( p_u \) to be combined with node \( u \)'s services, are those including service \( s \) at position \( i \)); and a domination check set \( dc(u, p_u) \subset ins(p_u + u) \) specifying what optimal instances of path \( p_u + u \) become unavailable, thus, when updating the optimal instances of path \( p_u + u \), all its instances previously dominated by at least one instance in \( dc(u, p_u) \) should be checked for optimality.

Algorithm 1 summarises the repair process of the optimal instances of path \( p_u + u \) at node \( u \), according to above semantics. Procedure \( \text{check-instance-optimality}(ins, u, p_u) \) assesses the optimality of instance \( ins \) against those already recorded at node \( u \) for path \( p_u + u \). Note that \( as(u, p_u) = ai(u, p_u, i \in \mathbb{Z}^+) = dc(u, p_u) = \emptyset \), when no modifications to the optimal instances of path \( p_u + u \) are required. Once the repair for the nodes is completed, the new best solution is the one with the highest utility among the adjusted optimal instances at node \( t_{inv} \) (the current end node).

The instantiation of sets \( as, ai, \) and \( dc \), for each valid predecessor, depends on the change type, as defined next.
Algorithm 1 repair-optimal-instances\((u, p_u)\)

1: \(v \leftarrow e\text{n}(p_u)\)
2: \(p_v \leftarrow p_u - v\)
3: \(\text{if } dc(u, p_u) \neq \emptyset\) then
4: \(\text{for each } s \in r\text{cnd}(u) \setminus a\text{s}(u, p_u) \text{ do}\)
5: \(\text{for each } ins_v \in o\text{i}(v, p_v) \text{ s.t. } \exists i \in \mathbb{Z}^+, sai(ins_v, i) \notin ai(u, p_u, i) \text{ do}\)
6: \(\text{if } \exists ins \in dc(u, p_u), ins \text{ } r\text{-d}m \text{ } ins_v + s \text{ then}\)
7: \(\text{check-instance-optimality}(ins_v + s, u, p_u)\)
8: \(\text{if } \exists i \in \mathbb{Z}^+, ai(u, p_u, i) \neq \emptyset \text{ then}\)
9: \(\text{for each } ins_v \in o\text{i}(v, p_v) \text{ s.t. } \exists i \in \mathbb{Z}^+, sai(ins_v, i) \in ai(u, p_u, i) \text{ do}\)
10: \(\text{if } \forall a \in AR, cr(ins_v + s, a) \text{ is-better-than } rc(a) \text{ then}\)
11: \(\text{check-instance-optimality}(ins_v + s, a, p_u)\)
12: \(\text{if } a\text{s}(u, p_u) \neq \emptyset \text{ then}\)
13: \(\text{for each } s \in a\text{s}(u, p_u) \text{ do}\)
14: \(\text{for each } ins_v \in o\text{i}(v, p_v) \text{ do}\)
15: \(\text{if } \forall a \in AR, cr(ins_v + s, a) \text{ is-better-than } rc(a) \text{ then}\)
16: \(\text{check-instance-optimality}(ins_v + s, a, p_u)\)
17: \(a\text{s}(u, p_u) \leftarrow \emptyset; a\text{i}(u, p_u, i \in \mathbb{Z}^+) \leftarrow \emptyset; dc(u, p_u) \leftarrow \emptyset\)

Procedure 2 check-instance-optimality\((ins, u, p_u)\)

1: \(\text{optml} \leftarrow 1\)
2: \(\text{for each optimal instance } ins_u \in o\text{i}(u, p_u) \text{ do}\)
3: \(\text{if } ins_u \text{ r-dm } ins \text{ then}\)
4: \(\text{optml} \leftarrow 0\)
5: \(\text{break}\)
6: \(\text{else if } ins \text{ r-dm } ins_u \text{ then}\)
7: \(\text{remove } ins_u \text{ from the instances at } o\text{i}(u, p_u)\)
8: \(\text{if } \text{optml}=1 \text{ then}\)
9: \(\text{add } ins \text{ to the instances } o\text{i}(u, p_u)\)

7.2.1. Addition of a service

Where a new service \(s_n\) joins the services of node \(t_{ch} \in V_{PK}\), sets \(as, ai,\) and \(dc\) of each valid predecessor \(p_u \in v\text{l}d\text{prd}(u \in V_{PK})\) are updated as follows. If \(p_u\) is a newly added valid predecessor as a result of this change, or \(u = t_{inv}\), the optimal instances of path \(p_u + u\) need to be recomputed, and thus set \(a\text{s}(u, p_u)\) should be assigned all the request-based non-dominated services of node \(u\), i.e. \(a\text{s}(u, p_u) = r\text{cnd}(u)\). Otherwise,
the following three cases are distinguished. Case 1: $p_u$ is not affected by this addition, i.e. $(u \neq t_{ch}) \land (t_{ch} \notin \text{nodes}(p_u))$. In this case, no change is made to the sets $as$, $ai$, and $dc$ associated with $p_u$. Case 2: $p_u$ is a valid predecessor of node $t_{ch}$, i.e. $u = t_{ch}$. In this case, only set $as(u, p_u)$ is modified, by adding to it the new service $s_i$ while removing all the services belonging to $RM$, i.e. $as(u, p_u) \leftarrow (as(u, p_u) \setminus RM) \cup AD$. In addition, all existing optimal instances ending with a service in $RM$ are eliminated, i.e. $oi(u, p_u) \leftarrow oi(u, p_u) \setminus \{ins \in oi(u, p_u) \mid es(ins) \in RM\}$. Case 3: $p_u$ is a valid predecessor including node $t_{ch}$ at position $i$, i.e. $io_n(p_u, t_{ch}) = i$. In this case, only set $ai(u, p_u, i)$ is modified, by adding to it the new service $s_i$ while removing all the services that are members of $RM$, i.e. $ai(u, p_u, i) \leftarrow (ai(u, p_u, i) \setminus RM) \cup AD$. Like the previous case, all the optimal instances including a service from $RM$ at position $i$ are eliminated, i.e. $oi(u, p_u) \leftarrow oi(u, p_u) \setminus \{ins \in oi(u, p_u) \mid sai(ins, i) \in RM\}$.

7.2.2. Deletion of a service

Where a service $s_o$ of node $t_{ch} \in V_{PK}$ becomes unavailable, sets $as$, $ai$, and $dc$ of each valid predecessor $p_u \in vldprd(u \in V_{PK})$ are updated as follows. If $u = t_{inv}$, the optimal instances of path $p_u + u$ need to be recomputed, and thus set $as(u, p_u)$ should be assigned the request-based non-dominated services of node $u$ (i.e. the instance already executed $ins_{exc} + s_{invo}$). Otherwise, the following five cases are distinguished. Case 1: $p_u$ is not affected by this deletion, i.e. $(u \neq t_{ch}) \land (t_{ch} \notin \text{nodes}(p_u))$. In this case, no change is made to sets $as$, $ai$, and $dc$ associated with $p_u$. Case 2: $p_u$ is a valid predecessor of node $t_{ch}$, i.e. $u = t_{ch}$, and the eliminated service $s_o \in as(u, p_u)$ (i.e. $p_u$ has not been reprocessed yet following the addition of service $s_o$ previously).

In this case, only set $as(u, p_u)$ is modified, by adding to it all the services in $AD$ while eliminating the deleted service $s_o$, i.e. $as(u, p_u) \leftarrow (as(u, p_u) \setminus RM) \cup AD$. Case 3: $p_i$ is a valid predecessor of node $t_{ch}$ and service $s_o \notin as(u, p_u)$. In this case, all the existing optimal instances ending with $s_o$, $IE = \{ins \in oi(u, p_u) \mid es(ins) \in RM\}$, are eliminated from $oi(u, p_u)$, i.e. $oi(u, p_u) \leftarrow oi(u, p_u) \setminus IE$, and added to set $dc(u, p_u)$, i.e. $dc(u, p_u) \leftarrow dc(u, p_u) \cup IE$. Case 4: $p_u$ is a valid predecessor including node $t_{ch}$ at position $i$, i.e. $(t_{ch} \in \text{nodes}(p_u)) \land (io_n(p_u, t_{ch}) = i)$, and the eliminated service $s_o \in ai(u, p_u, i)$ (i.e. $p_u$ has not been reprocessed yet following the addition
of service \( s_o \) previously). In this case, only set \( ai(u, p_u, i) \) is modified, by adding to it all the services in \( AD \), while eliminating the deleted service \( s_o \), i.e. \( ai(u, p_u, i) \leftarrow (ai(u, p_u, i) \setminus \text{RM}) \cup AD \). Case 5: \( p_u \) is a valid predecessor including node \( t_{ch} \) at position \( i \) and service \( s_o \notin ai(u, p_u, i) \). In this case, all the existing optimal instances containing service \( s_o \) at position \( i \), \( IAI = \{ \text{ins} \in oi(u, p_u) \mid sai(ins, i) \in \text{RM} \} \), are eliminated from \( oi(u, p_u) \), i.e. \( oi(u, p_u) \leftarrow oi(u, p_u) \setminus IAI \), and added to the set \( dc(u, p_u) \), i.e. \( dc(u, p_u) \leftarrow dc(u, p_u) \cup IAI \).

### 7.2.3. Changes in the quality values of a service

Changes in the quality values of a service \( s_o \) of node \( t_{ch} \in V_{PK} \), with \( s_{ch} \) denoting this service after the change, can be defined in terms of the deletion and addition of a service, as follows. If service \( s_o \) r-dm \( s_{ch} \), or service \( s_{ch} \notin AD \), this case is modelled as the deletion of service \( s_o \) with \( AD_{del} = AD \), \( RM_{del} = \text{RM} \). Therefore, the same updates to the sets \( as \), \( ai \), and \( dc \) in the deletion case are applied here. Otherwise, this case is treated similarly to the deletion of a request-based non dominated service \( s_o \) with \( AD_{del} = AD \setminus \{ s_{ch} \} \), \( RM_{del} = \{ s_o \} \), followed by a subsequent addition of service \( s_{ch} \) with \( AD_{add} = \{ s_{ch} \} \), \( RM_{add} = \text{RM} \setminus \{ s_o \} \).

### 7.2.4. Example

Suppose that during the execution of \( s_{B1} \) of the initial solution \( s_{B1}s_{C2}s_{E2}s_{F1} \) of Section 4, service \( s_{E3}(10, 50) \) joins the services of node \( E \). This change is a change to be considered with \( AD = \{ s_{E3} \} \), \( RM = \{ \emptyset \} \), and no effect on the valid predecessors of nodes. Given the search graph of Figure 7, the instantiation of sets \( as \), \( ai \), and \( dc \), for the valid predecessors, in response to this change, is provided in Figure 8.

### Figure 8: Sets as, ai, and dc in the running example

<table>
<thead>
<tr>
<th>Node</th>
<th>( F )</th>
<th>( E )</th>
<th>( D )</th>
<th>( C )</th>
<th>( B )</th>
<th>( A )</th>
<th>( G )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{valid} )</td>
<td>( S )</td>
<td>( SF )</td>
<td>( \emptyset )</td>
<td>( SFE )</td>
<td>( SFEC )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>as</td>
<td>( \emptyset )</td>
<td>( { s_{E3} } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( { s_{B1} } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>ai</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( { (s_{E3}, \text{index} = 2) } )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>dc</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
8. Analytical Study

This section analyses the time complexity of the proposed repair-based re-selection algorithm, and compares it against re-selection from scratch (also assumed to be applied on the reverse version of the plan paths graph for ease of comparison). We focus here on the step of optimal instances modification in response to a change since, when compared to this step, the time required for the other change handling steps (e.g. updating the request-based non-dominated services of affected nodes and identifying the change category) is negligible.

Since the valid predecessors at each node are processed independently, the analysis assumes for simplicity one valid predecessor per node (the specific case of one abstract plan). This still allows demonstration of the efficiency gain achieved by our approach for any affected valid predecessor, and can easily be generalised to handle the case of multiple valid predecessors (i.e. multiple abstract plans). Note that, in such a general case, our approach achieves further time reduction due to reprocessing only the affected valid predecessors per each node, compared to the re-selection from scratch which reprocesses all the valid predecessors.

Next, the time of the pre-execution selection algorithm is analysed in order to provide the basis for the subsequent analysis of re-selection approaches.

8.1. Selection Algorithm

Consider a sequential abstract plan comprising $k$ tasks, $v_1v_2...v_k$, each with $n$ available candidate services. To select the best solution (the best instance of path $v_1v_2...v_k$), each node $v_{i>1}$ records the optimal instances of path $v_1v_2...v_i$, denoted $oi(v_i)$. Hence, selection time $\tau(sel)$ is:

$$\tau(sel) = \sum_{i=2}^{k} \tau(oi(v_i))$$  \hspace{1cm} (1)

The time required to calculate $oi(v_i)$, $\tau(oi(v_i))$, depends on the sizes of $oi(v_{i-1})$ and $rcnd(v_i)$, which can be defined in terms of the following pruning rates: $spr_i \in [0, 1]$, denoting the percentage of candidate services pruned per task node $v_i$ prior to selection, i.e. $|rcnd(v_i)| = n \times sr_i$, where $sr_i = 1 - spr_i$; and $cpr_i \in [0, 1]$, denoting
the percentage of instances pruned per path \(v_1v_2...v_i\) when computing the optimal instances at node \(v_i\), i.e. \(|oi(v_i)| = |oi(v_{i-1})| \times |rcnd(v_i)| \times cr_i\), where \(cr_i = 1 - cpr_i\).

Since \(|oi(v_1)| = |rcnd(v_1)| = n \times sr_1\),

\[
|oi(v_i)| = n^i \times \prod_{m=1}^{i} sr_m \times \prod_{m=2}^{i} cr_m
\]  

(2)

Based on this, \(\tau(oi(v_{i>1}))\) is given as follows:

\[
\tau(oi(v_i)) \in O((|oi(v_{i-1})| \times |rcnd(v_i)|)^2) \\
\text{is } O(n^{i-1} \times \prod_{m=1}^{i-1} sr_m \times \prod_{m=2}^{i-1} cr_m \times n \times sr_i)^2) \\
\text{is } O(n^{2i} \times \prod_{m=1}^{i} sr_m^2 \times \prod_{m=2}^{i-1} cr_m)
\]  

(3)

From Equations 1 and 3, the time complexity of selection \(\tau(\text{sel})\) is

\[
\tau(\text{sel}) \in O(n^{2k} \times \prod_{m=1}^{k} sr_m^2 \times \prod_{m=2}^{k} cr_m)
\]

Note here that \(k \ll n\).

Assuming for simplicity that \(\forall i, sr_i = cr_i = r\), \(\tau(\text{sel})\) is \(O(n^{2k} \times r^{4k-4})\). Hence, our service selection achieves a time complexity of \(O(n^\alpha)\) if rate \(r = \frac{4k-4}{\sqrt{n^\alpha}-2k}\).

For example, in order for selection to be of linear time complexity, i.e. \(\alpha = 1\), when \(n = 100\) and \(k = 5\), rate \(r\) should be: \(r = \sqrt{100} = 0.08\). That is, the pruning rate \((1-r)\) should be at least 92%.

8.2. Reselection from scratch

The re-selection from scratch approach recalculates the optimal instances of all non-executed nodes from scratch, in response to a change to be considered at node \(v_{ch}\) while executing node \(v_{inv}\). Thus, its time complexity, \(\tau^s(\text{resel})\), is:

\[
\tau^s(\text{resel}) = \sum_{i=1}^{in_v-1} \tau^s(oi_n(v_i))
\]  

(4)

Here, \(\tau^s(oi_n(v_i))\) is the time required for recomputing the optimal instances at node \(v_i\), given as (see Equation 3):

\[
\tau^s(oi_n(v_{i<ch})) \in O(n^{2i} \times \prod_{m=1}^{i} sr_m^2 \times \prod_{m=2}^{i-1} cr_m)
\]

\[
\tau^s(oi_n(v_{i\geq ch})) \in O(n^{2(i-1)} \times n^2 \times \prod_{m=1}^{i-1} sr_m^2 \times \prod_{m=2}^{i-1} cr_m)
\]
where \( n' = |\text{chd}_{n}(v_{ch})| \), i.e. \( n' = n + 1 \) in case of service addition; \( n' = n - 1 \) in case of service deletion; and \( n' = n \) in case of changes in service qualities. Based on this and Equation 4, we can conclude that \((inv \leq k)\):

\[
\tau^s(\text{resel}) = O(n^{2(inv-1)} \times \prod_{m=1}^{inv-1} sr_m^2 \times \prod_{m=2}^{inv-2} cr_m^2) \tag{5}
\]

8.3. Repair-based Reselection

The proposed repair-based re-selection approach only makes the updates necessary to the affected optimal instances, in response to a change to be considered at node \( v_{ch} \) while executing node \( v_{inv} \), without recalculating those instances from scratch. Thus, its time complexity, \( \tau^r(\text{resel}) \), is:

\[
\tau^r(\text{resel}) = \sum_{i=ch}^{inv-1} \tau^r(oi_n(v_i)) \tag{6}
\]

Here, \( \tau^r(oi_n(v_i)) \) is the time required for modifying the optimal instances at node \( v_i \).

The modification depends on the type of change that has occurred, and is analysed next for the addition case, i.e. addition of a request-based non-dominated service \( s_n \) at node \( v_{ch} \) (the deletion and quality changes cases can be analysed similarly).

For node \( v_{ch} \), the modification involves combining the optimal instances of node \( v_{ch-1} \) with service \( s_n \), and then checking the optimality of the resulting combinations against those already recorded at node \( v_{ch} \), i.e. \( \tau^r(oi_n(v_{ch})) \) is:

\[
O(|oi_n(v_{ch-1})| \times |oi_n(v_{ch}))
\]

is \( O(n^{ch-1} \times \prod_{m=1}^{ch-1} sr_m \times \prod_{m=2}^{ch-1} cr_m \times n^{ch} \times \prod_{m=1}^{ch} sr_m \times \prod_{m=2}^{ch} cr_m) \)

is \( O(n^{2ch-1} \times \prod_{m=1}^{ch-1} sr_m^2 \times \prod_{m=2}^{ch-1} cr_m^2 \times sr_{ch} \times cr_{ch}) \) \tag{7}

Similarly, for node \( v_{i>ch} \), updating \( oi_{n}(v_i) \) involves checking the optimality of the newly available instances (obtained by joining the optimal instances containing service \( s_n \) at node \( v_{i-1} \), \( oi_n(v_{i-1})^{s_n} \), with node \( v_i \)'s services), against those already computed
at node $v_i$, i.e. $\tau^r(o_{in}(v_i>ch))$ is:

$$O(|o_{in}(v_{i-1})| \times |rend(v_i)| \times |o_{in}(v_i)|)$$

is $O\left(\frac{n^{i-2} \times n' \times \prod_{m=1}^{i-1} sr_m \times \prod_{m=2}^{i} cr_m}{n' \times sr_{ch}} \times n \times sr_i \times \prod_{m=2}^{i} cr_m\right)$

is $O\left(n^{i-1} \times n' \times \prod_{m=1}^{i} sr_m \times \prod_{m=2}^{i} cr_m\right)$

is $O\left(n^{2(i-1)} \times n' \times \prod_{m=1}^{i} sr_m^2 \times \prod_{m=2}^{i} cr_m^2 \times sr_{ch} \times cr_i\right)$

(8)

Here, $n' = n + 1$, and $|oi(v_i)| = \frac{|oi(v_i)|}{|rend(v_i)|}$. From Equations 6, 7 and 8, we conclude that $\tau^r(resel)$ is:

$$O(n^{2i-3} \times \prod_{m=1}^{inv-1} sr_m^2 \times \prod_{m=2}^{inv-2} cr_m^2 \times sr_{ch} \times cr_{inv-1})$$

(9)

This is applicable as long as $ch < inv$ (the node affected by the change is not the node being executed). Yet, when $ch = inv$ (the invoked service delivers unexpected qualities), reselection only involves recombining the optimal instances already recorded at node $v_{inv-1}$ with the modified invoked instance, and thus $\tau^r(resel)$ is $O(|oi_o(v_{inv-1})|)$, i.e.

$$\tau^r(resel) = O(n^{inv-1} \times \prod_{m=1}^{inv-1} sr_m \times \prod_{m=2}^{inv-1} cr_m)$$

(10)

8.4. Comparison

To analyse the efficiency gain achieved by the proposed repair-based re-selection (compared to reselection from scratch), we make the simplifying assumption that $\forall i$, $sr_i = cr_i = r$. As a result, comparing time complexities $\tau^s$ and $\tau^r$, leads to:

$$\frac{\tau^s(resel)}{\tau^r(resel)} = \frac{n^{2inv-2} \times r^{4inv-8}}{n^{2inv-3} \times r^{4inv-8}} = n$$

$$\frac{\tau^s(resel)}{\tau^r(resel)} = \frac{n^{inv-1} \times r^{2inv-5}}{n^{inv-1} \times r^{2inv-3}} = 1$$

In other words, when the change occurs at a non-executed node, the proposed approach reduces reselection time by a factor of $n$. The reduction factor further increases to
in the case where the change affects the node being executed (i.e. cannot be anticipated in advance).

9. Empirical Study

The goal of this section is to assess the efficiency of our repair-based re-selection algorithm, the gain in utility by responding to changes ahead of time, and the reduction in execution interruption by performing re-selection without interfering with the execution process (unless necessary).

9.1. Experimental Setup

We perform the evaluation in the domain of learning object composition [30], where the goal is to fulfill a particular learning objective by automatically compositing existing reusable learning objects (the candidate services in our model) into a respective course, taking learner (user) preferences and constraints into consideration (full details on this case study can be found elsewhere [31]). Learning objects (LOs) are published through learning object repositories, where hundreds of learning objects with different properties can be available for each concept, and are usually heterogeneous in their granularities, i.e. they can range from a single image to a whole module. Thousands of new learning objects are made available every day, while existing learning objects can be updated or become unavailable at any time. Such changes in the repository can occur during the delivery (execution) of a selected course (a composition of learning objects), thus possibly necessitating the re-selection of learning objects for concepts not yet presented, in order to guarantee the most suitable learning experience for the user.

The metadata elements (quality of service properties) of 36956 learning objects were collected from 10 different repositories, using the OAI-PMH protocol. Of those elements, we selected the following for the global-level constraints and utility function: interactivity type, semantic density, difficulty, typical learning time, size, and cost. The planning knowledge adopted is a hierarchical representation of the Algorithms and

[http://www.openarchives.org/OAI/openarchivesprotocol.html]
Data Structure domain, where the tasks correspond to domain concepts. It comprises 3 hierarchical levels, with up to 11 concepts (tasks) per abstract plan in the search graph.

Learning objects are examples of services with long execution durations $\text{execTime}$ (corresponding to learning time). For such long-running services, where $\text{execTime} \gg \text{rslScratch}$ ($\text{rslScratch}$ is the time anticipated to perform the most costly re-selection from scratch, for the non-executed tasks), the proposed continuous adaptive behaviour may cause an unnecessary overhead from the composite service provider perspective. This is because, when a service execution spans a long period of time $[t_s, t_e]$, it becomes unnecessary to continuously react to changes for this entire duration. Instead, all changes could be ignored until $t_{\text{crt}} = t_e - \text{rslScratch}$, at which point re-selection from scratch for the remaining tasks should be instantiated to restore a valid instance of the search graph with respect to the new environment state. From this point, the execution should continue in the proposed light repair-based manner, efficiently accommodating all the changes occurring during the critical interval $[t_{\text{crt}}, t_e]$ (the interval signaling the end of the current service execution). This adaptation of the execution behaviour saves the cost of maintaining the search graph and triggering re-selection a potentially very large number of times. Given this, only the critical short interval $[t_{\text{crt}}, t_e]$ is relevant for the purpose of evaluating our approach, and therefore we assume next a short execution time per learning object, ignoring the irrelevant long interval $[t_s, t_{\text{crt}}]$. This simplifies the experiments and facilitates averaging the results over multiple runs. All the results reported are averaged over 30 randomly-generated requests (with two global-level constraints and a utility optimisation requirement).

9.2. Experimental Results

To assess the gain in performance obtained by the repair-based re-selection approach, the time required for re-selecting services for the remaining tasks in response to an affecting change at execution time is compared in two cases: where re-selection from scratch is applied, and where repair-based re-selection is applied. As explained in Section 7, reacting to a change in the former case requires recalculating the optimal instances of the non-executed nodes from scratch, while in the latter case, the adaptation process is achieved by only making the necessary updates to the optimal instances.
Figure 9: Reselection time in response to a random change

Figure 10: Reselection time in response to a violation in the executed LO’s qualities
already recorded at nodes. Here, the number of learning objects (i.e. services) considered per task is fixed at 500, while the execution position (the index of the learning object being executed) when the change occurs is varied between 1 and 10 (11 is the total number of tasks per abstract plan). Figures 9 and 10 compare the two cases in terms of running time, averaged over a number of different random requests. In Figure 9, change types (addition, deletion, or changes in qualities) and locations (the tasks and services affected) at each execution position are selected randomly, whereas those considered in Figure 10 are receiving unexpected quality values from the executed services. As can be seen, the repair-based re-selection significantly outperforms the re-selection from scratch, especially when the change is discovered at an early stage of execution. Moreover, both cases require less time with the increasing execution position. This is because, as more services are executed, the number of graph nodes to be considered in the re-selection process decreases, and so does the number of their optimal instances. We can also observe from the situation studied in Figure 10 in which it is not possible to perform the adaptation process in parallel with execution since the erroneous behaviour cannot be discovered prior to its occurrence, that almost no interruption in execution will be caused by the repair-based approach proposed.

To assess utility gain from the early reaction to changes, the solution optimality
Figure 12: Changes in the qualities of the selected learning object at position 11

Figure 13: Deletion of a selected learning object (Exec. Pos. =1)
obtained as a result of reacting to a change in the selected learning objects as soon as it occurs in the environment is compared with that of the delayed reaction (i.e. when the unavailable learning object is invoked or after the quality violating learning object is executed). This optimality is estimated as $\frac{c_{\text{act}}}{c_{\text{opt}}}$, where $c_{\text{act}}$ is the actual utility achieved by re-selecting services for the non-executed tasks, and $c_{\text{opt}}$ is the optimal utility assuming no task is executed (i.e. the utility of the best solution according to the current environment state, and given that no task is executed). Figures 11 and 12 show the results in the cases where the last service in the selected solution becomes unavailable, and changes its qualities, respectively, varying the execution position at which the change occurs (i.e. at which the re-selection is triggered by the early approach to change handling) between 1 and 10 (each solution composite service is comprised of 11 services). As expected, the earlier change adaptation is performed, the better the utility of the resulting solution, which emphasises the importance of responding to changes as early as possible. This is further highlighted in Figures 13 and 14, where the execution position at which the change is observed is fixed at 1 (i.e. the change occurs while executing the first service of the selected solution), while the change location (the index of the task affected by the change) ranges between 2 and 11. Clearly, the optimality achieved by the delayed re-selection decreases as more services become
Finally, the last part of the experiments evaluates the reduction in interruption time between component service executions, achieved as a result of reacting to changes in the environment as soon as they occur, in parallel with the execution of the current component service. This parallelism is simulated using multi-threading on the composite service provider side, with the execution of a component service being simulated by invoking a service on a remote computer, which simply sleeps for a certain amount of time \( \text{execTime} \) (service execution time) before returning a result. Changes occurring during execution are generated randomly in the interval \( [\text{start}, \text{end} = \text{start} + (\text{execTime} \times \text{maxNum})] \), where \( \text{start} \) is the start time of the composite service execution, while \( \text{maxNum} \) is the maximum number of component services (learning objects) in a composite solution. The type of each change (addition, deletion, or changes in qualities), and its location (the task and the learning object affected by the change) are also selected randomly. Figure 15 shows the delay time after completing the execution of each learning object in the composite solution, averaged over a number of different runs. In each run, 20 changes to be considered are introduced during each component service execution, while the number of services per task, service execution time \( \text{execTime} \), and the number of tasks per abstract plan \( \text{maxNum} \), are fixed.
Figure 16: The effect of the number of changes per learning object execution

...at 500, 5 seconds, and 11 tasks, respectively. The results indicate that, even with this large number of changes, the interruption time achieved by our approach is small (i.e., the re-selection process in response to the changes is almost completed before the current component service finishes its execution), especially with the increasing execution position (the re-selection process requires less time when more component services become executed, due to the decreased number of nodes to be considered in the re-selection process).

The interruption time is further evaluated in Figures 16 and 17 with respect to service execution time $\text{execTime}$. Figure 16 shows the interruption time between the first and second component service executions, varying the number of changes introduced during the first service execution between 5 and 25 (all the changes are assumed to be interrupting). As expected, the interruption time decreases with the decreasing number of changes and the increasing service execution time. Figure 17, on the other hand, shows the effect of varying the time slot within which the change occurs, on interruption time. For this purpose, the service execution time $\text{execTime}$ is divided into 25 equal time intervals, during which an interrupting change is introduced while executing the first component service. Intuitively (as shown in Figure 17), the earlier the change occurs during a service execution, the more likely that no interruption...
will be caused by the corresponding re-selection.

10. Conclusion

In this paper, we have presented a novel adaptive execution algorithm, capable of handling execution-time service changes for both repairing and optimisation purposes. To achieve a light adaptation process, the algorithm reuses the optimal instances generated during the selection process. For this purpose, it assumes a reverse version of the search graph, which allows response to changes by applying only a minimal number of modifications to the graph, without the need to perform re-selection from scratch. The adaptation process is triggered as soon as changes occur in the environment, without interfering with the execution process, unless necessary. This need is identified based on a categorisation of changes, specifying their urgency and importance, and guiding the behaviour of the executing system. Via such an early, parallel-to-execution, and light reaction approach, the chances of a successful recovery are maximised and solution optimality is increased, while reducing execution disruptions as much as possible, as demonstrated through the evaluation conducted. The results also show that, even in the cases where interference with execution is non-preventable (e.g. when an executed
service delivers unanticipated quality values), the algorithm manages to recover from the situation with minimal interruption.

References


